VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ STUDIJŲ PROGRAMA

Modern C++ for Data Structures and Algorithms

Duomenų Struktūros ir Algoritmai Moderniame C++

Kursinis darbas

Author:	4 kurso I grupės studentas
	Domas Kalinauskas
Supervisor:	Viktoras Golubevas

Vilnius – 2024

Contents

Įvadas/Intro?

0.1. Tikslas/Aim

The aim of this <Kursinis darbas> is to analyze how modern C++ improves on the usability of DSA's (data structures and algorithms) and their performance, in comparison to older versions of the C++ standard, and how it stacks up with other languages.

0.2. Uždaviniai/Objectives

1. 2. 3.

0.3. Ivadas/Intro

Collectively, the world is moving towards safer and safer languages. This is seen via widespread adoption of memory-safe languages, such as Rust, or the overall popularity of 'managed', garbage-collected runtimes such as Java, .NET, Python etc. (The safety in 'managed' languages comes from the fact that a vulnerability in the runtime has a much smaller surface area – requiring only a fix in the runtime, instead of the code that is built on top of it)

However, C++ has seen somewhat of a 'renessaince', with a similiar move to safer and more declarative APIs, along with a push towards 'functional' style concepts. This <Kursinis darbas> aims to be a comprehensive comparison-analysis of recently introduced modern C++ features, and how they stack up against similar features in other languages. We'll <galiu naudot?> take a look at the memory impact, performance impact, and implementation details of these features, along with where they might be ideally used.

Įvade apibūdinamas darbo tikslas, temos aktualumas ir siekiami rezultatai. Darbo įvadas neturi būti dėstymo santrauka. Įvado apimtis 1—2 puslapiai.

1. Modern C++ Features for Algorithmic Problem Solving

1.1. Ranges

A common pattern in C++ code is applying an operation over a selection of elements. The most classic style, which is still in-use today is the index based loop:

```
void operate(std::span<uint32_t> values) {
    for (size_t i = 0; i < values.size(); ++i) {
        std::print("Value: {}", values[i]);
    }
}
```

There are a few notable downsides to the index based for loop, namely that it's error-prone, and can only be effectively used with random-indexable types (e.g. std::array, std::vector). This means that if we want to iterate over a list, or another custom container, we'd have to change the for loop structure to be compatible.

With the introduction of C++11, we got access to the range-based for loop (*for production you'd want to use a specific concept instead of auto - that way you would get a clearer error message):

```
void operate(const auto &container) {
    for (const auto& x : container) {
        std::print("Value: {}", x);
    }
}
```

When using range-based for loops, we no longer have to manually write the iteration code, meaning it doesn't matter if the type is a array, vector, list, or any other custom iterator. Under the hood, these range-based for loops relies on the container having .begin()/.cbegin() and .end()/.cend() member functions - they return corresponding iterators which allow access to values.

These iterators are what is passed along to the STL algorithms, however, they don't allow for full freedom - they become quite verbose when combining more complex operations.

Take for example, we had a scenario where we're given a collection of numbers, we want to skip the first N values, filter out the even ones, then apply some operation on them, and finally print them out - all while quitting early, if we encounter some magic number.

```
static double transform_number(int input);
static void print(double val);
void operate(const auto &container, size_t offset, double early_exit) {
    /* Skip first 3 elements (container.begin() + offset isn't supported for all iterato
    std::find_if(std::next(container.begin(), offset), container.end(), [&](int val) {
        /* Filter only for odd values*/
        if (!is_odd(val)) {
            return false;
        }

        /* Apply operation */
        double transformed = transform_number(val);

        /* Check if early exit encountered */
        if (transformed == early_exit) {
            return true;
        }

        /* Else - print value & continue */
        print(transformed);
        return false;
    });
}
```

This is quite verbose - We have to use std::next to ensure support for (most) iterator types, pass the begin and end manually, use std::find_if (keeping in mind that false means to continue iterating - normally you'd expect the inverse). This isn't hard to understand, but it's certainly not clear, especially when someone might not know exactly what is going on.

This becomes much simpler when we use ranges, introduced first in C++20. With ranges and views(*special type of range, where the operations are lazy), the same functionality can be implemented in a much simpler way:

```
static double transform_number(int input);
static void print(double val);
void operate(const auto &container, size_t offset, double early_exit) {
    auto elems = container
        | std::views::drop(offset)
        | std::views::filter(is_odd)
        | std::views::transform(transform_number)
        | std::views::take_while([&](auto v){return v != early_exit;});
    std::ranges::for_each(elem,print);
}
```

However, C++ ranges suffer from some less than obvious safety issues and performance downsides (*reference talk about C++ ranges filter). One of which, being a very less than obvious footgun(! const propogation - filter re-use after modification)

Let's take a look at what gets executed when we call that function, while printing when performing drop, filter, transform and take_while.

```
static double transform_number(int input) {
    return input * 5;
}

int main() {
    static constexpr auto nums = std::array{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    operate_ranges(nums, 3, 45.0);
    return 0;
}
```

The output for execution looks like this:

```
drop
drop
drop
drop /* This drop 'opens' the flood gates */
filter /* Filter out even value */
filter /* Value passes filter*/
transform /* Value is transformed */
take /* We execute take */
transform /* Value is taken and then... transformed? */
25.000000 /* Value is printed */
filter /* ... continue as above */
filter
transform
take
transform
35.000000
filter
filter
transform
take /* Value doesn't pass take - early exit condition */
```

The unexpected part here is that we have to apply transformation twice for each value passing filter - once when evaluating whether value is taken, and then again when we dereference it. This is because, semantically, in the C++ iterator model, positioning (++) and accessing (*) are distinct operations. Under the hood, take must access (*) the value, then change positioning (++) of the iterator. After the value is taken, then our for_each does the same thing - it dereferences whatever we get out, meaning that the transformation gets applied twice, since there's no way of 'reusing' the accessed value.

This differs from, for example, the Rust iterator model, where essentially each stage of the pipeline passes values directly to the next. This becomes apparent when you want to implement your own Rust iterators vs C++ views. In the C++ case, the view takes some input view, and then must define both a begin() and an end() operation which return 'iterators', whereas with the Rust variant, only a next() function, returning an Optional<ValueT> is required.

Here is that same example, written using Rust iterators, along with the produced output:

```rust
fn operate_ranges<T>(
    container: T,
    offset: usize,
    early_exit: f64,
) where
    T: IntoIterator<Item = i32>,
{
    container.into_iter()
        .skip(offset)
        .filter(|&v| { is_odd(v) })
        .map(|v| { transform_number(v) })
        .take_while(|&v| { v != early_exit})
        .for_each(|v| print(v));
}
```

```
drop
drop
drop
drop
filter
filter
transform
take /* Up to here, matches C++ */
/* We see here, that when using the Rust model -
transformation doesn't need to be applied again */
25
filter
filter
transform
take
35
filter
filter
transform
take
```

Duplicate work can be avoided in the C++ case, by using a view that caches the result of dereferencing an element, but the downside is that, depending on what it is you're caching, it can become quite expensive. Unfortunately, while such a view was present in ranges-v3(*link) which was used to form the basis of

P0896R4(*link)(the proposal that got accepted into the C++20 standard), quite a few of the views in ranges-v3 were missing, views::cache1 being one of them.

FIXME: Cia reiktu kokio normalus conclusion'o? Kazka daugmas kad C++20+ ranges padidinima kodo skaitomuma, bet performance-sensitive vietose geriau jo vengti.

1.2. Concepts

1.3. Functional

(Variants, Optional, Expected, Transform, t.t.)

Compare with equivalent features in Rust (mixed-paradigm language) & Haskell (functional language)

2. Modern C++ for High-Performance Computing

2.1. Compile-time calculations

* Compare with other languages that can do that (Zig?) * Mention best compile-time LUT library $\hat{)}$ * Maybe something about compression at compile-time

2.2. Execution policies

* Reference TBB * Can be compared to Rust libraries with similiar thing (ehhh forgot but iterator)

2.3. Coroutines

* reference coroutine look-up? (https://www.youtube.com/watch?v=j9tlJAqMV7U) * compare to Rust async & maybe js? async

3. Medžiagos darbo tema dėstymo skyriai

Medžiagos darbo tema dėstymo skyriuose pateikiamos nagrinėjamos temos detalės: pradinė medžiaga, jos analizės ir apdorojimo metodai, sprendimų įgyvendinimas, gautų rezultatų apibendrinimas. Šios dalies turinys labai priklauso nuo darbo temos. Skyriai gali turėti poskyrius ir smulkesnes sudėtines dalis, kaip punktus ir papunkčius.

Medžiaga turi būti dėstoma aiškiai, pateikiant argumentus. Tekste dėstomas trečiuoju asmeniu, t.y. rašoma ne “aš manau”, bet „autorius mano“, „autoriaus nuomone“. Reikėtų vengti informacijos nesuteikiančių frazių, pvz., „...kaip jau buvo minėta...“, “...kaip visiems žinoma...” ir pan., vengti grožinės literatūros ar publicistinio stiliaus, gausių metaforų ar panašių meninės išraiškos priemonių.

Skyriai gali turėti poskyrius ir smulkesnes sudėtines dalis, kaip punktus ir papunkčius.

3.1. Poskyris

Citavimo pavyzdžiai: cituojamas vienas šaltinis [PPP01]; cituojami keli šaltiniai [Org00; Pav05a; Pav05b; PPP$^+$02; PPP03; PPŠ04; STU$^+$02; STU01; STU03; STU04; Sur05].

Anglų kalbos terminų pateikimo pavyzdžiai: priklausomybių injekcija (angl. *dependency injection*, dažnai trumpinama kaip *DI*), saitų redaktorius (angl. *linker*).

Išnašų[1] pavyzdžiai[2].

3.2. Faktorialo algoritmas

1 algoritmas parodo, kaip suskaičiuoti skaičiaus faktorialą.

Algorithm 1. Skaičiaus faktorialas

```
1: $N \leftarrow$ skaičius, kurio faktorialą skaičiuojame
2: $F \leftarrow 1$
3: for $i := 2$ to $N$ do
4:     $F \leftarrow F \cdot i$
5: end for
```

3.2.1. Punktas

3.2.1.1. Papunktis

3.2.2. Punktas

[1] Pirma išnaša.

[2] Antra išnaša.

4. Skyrius

4.1. Poskyris

4.2. Poskyris

Rezultatai ir išvados

Rezultatų ir išvadų dalyje turi būti aiškiai išdėstomi pagrindiniai darbo rezultatai (kažkas išanalizuota, kažkas sukurta, kažkas įdiegta) ir pateikiamos išvados (daromi nagrinėtų problemų sprendimo metodų palyginimai, teikiamos rekomendacijos, akcentuojamos naujovės).

Išvados

1. Išvadų skyriuje daromi nagrinėtų problemų sprendimo metodų palyginimai, siūlomos rekomendacijos, akcentuojamos naujovės.
2. Išvados pateikiamos sunumeruoto (gali būti hierarchinis) sąrašo pavidalu.
3. Darbo išvados turi atitikti darbo tikslą.

References

[Org00] Organizacijos Pavadinimas. Kodėl abėcėlė vadinasi ABC, o ne DEF? *Žurnalas*. 2000, volume I, pp. 1–20.

[Pav05a] A. Pavardonis. *Bakalauro darbo pavadinimas*. Vilnius, 2005. Bachelor's thesis. Universiteto pavadinimas.

[Pav05b] A. Pavardonis. *Magistrinio darbo pavadinimas*. 2005. Master's thesis. Universiteto pavadinimas.

[PPP$^+$02] A. Pavardenis, B. Pavardonis, C. Pavardauskas, D. Pavardinskas. Straipsnio pavadinimas. In: *Rinkinio pavadinimas*. Miestas, šalis: Leidykla, 2002, pp. 3–15.

[PPP01] A. Pavardenis, B. Pavardonis, C. Pavardauskas. Straipsnio pavadinimas. *Žurnalo pavadinimas*. 2001, volume IV, pp. 8–17.

[PPP03] A. Pavardenis, B. Pavardonis, C. Pavardauskas. *Knygos pavadinimas*. Miestas, šalis: Leidykla, 2003. 172 psl.

[PPŠ04] A. Pavardenis, B. Pavardonis, C. Šavardauskas. *Elektroninės publikacijos pavadinimas*. 2004. [visited on 2015-02-01]. Available from: `https://example.com/kelias/iki/straipsnio`.

[STU$^+$02] A. Surname, B. Tsurname, C. Usurname, D. Vsurname. Article title. In: *Conference book title*. City, country: Publisher, 2002, pp. 3–15.

[STU01] A. Surname, B. Tsurname, C. Usurname. Article Title. *Journal Title*. 2001, volume IV, pp. 3–15.

[STU03] A. Surname, B. Tsurname, C. Usurname. *Book title*. City, country: Publisher, 2003. 172 psl.

[STU04] A. Surname, B. Tsurname, C. Usurname. *Online Source Title*. 2004. [visited on 2015-02-01]. Available from: `https://example.com/path/to/the/article`.

[Sur05] A. Surname. *Title of PhD thesis*. London, 2005. PhD thesis. Title of university.

Santrumpos

Sąvokų apibrėžimai ir santrumpų sąrašas sudaromas tada, kai darbo tekste vartojami specialūs paaiškinimo reikalaujantys terminai ir rečiau sutinkamos santrumpos.

Appendixes

Appendix 1

Neuroninio tinklo struktūra

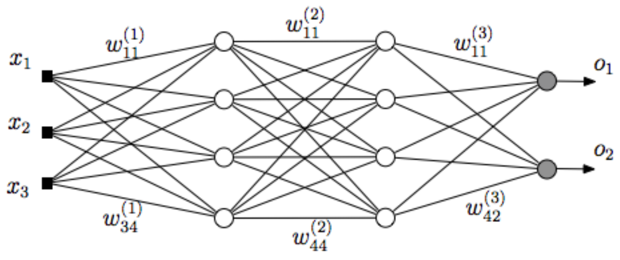

Figure 1. Paveikslėlio pavyzdys

Appendix 2

Eksperimentinio palyginimo rezultatai

Table 1. Lentelės pavyzdys

Algoritmas	$\bar{x}$	σ^2
Algoritmas A	1.6335	0.5584
Algoritmas B	1.7395	0.5647